

TopHat Question

[Anonymous]: How did you feel about the water skit last lecture and the raunchy elements of our skits in general?

- A. I don't want to see anything like that again; please tone it down
- B. I didn't really like it ; I'd prefer if you tone it down, but I am generally indifferent
- C. I feel neutral
- D. I thought it was a little funny
- E. I thought it was very funny

1 / 82

1

Revisiting Arrays: Size of 2-D Arrays

```
public static final int NUM_ROWS = 10; // defined in Constants
public static final int NUM_COLS = 5; // defined in Constants

public void practice2DArrays() {
    // deciding which is row and which is column index is
    // arbitrary but must be consistent!!!
    String[][] stringArray = new String[NUM_ROWS][NUM_COLS];
    int numRows = stringArray.length;
    int numCols = stringArray[0].length;
    System.out.println("My array has " + numRows * numCols + " slots in total!");
}
```

`array.length` gives size of first dimension (you decide whether you want row or column), and `array[0].length` gives size of second dimension

2 / 82

2

Common Array Errors - Watch Out! (1/2)

- Cannot assign a scalar to an array

```
int[] intArray = 5;
```



- 5 is not an array
- to initialize array elements, must loop over array and assign values at each index. Here we assign 5 to each element:

```
int[] intArray = new int[20]; //initializes array, not elements
for (int i=0; i < intArray.length; i++){
    intArray[i] = 5;
}
```

3 / 82

3

Common Array Errors - Watch Out! (2/2)

- Cannot assign arrays of different dimensions to each other

```
int[] intArray = new int[23];
int[][] 2DIntArray = new int[2][34];
intArray = 2DIntArray;
```



- Doing so will result in this error:

"Incompatible types: Can't convert int[] to int[][]"

- Similar message for assigning arrays of mismatched type
- Take note that Java will automatically resize an array when assigning a smaller array to a larger one

4 / 82

4

2D Arrays Example (1/2)

- Let's build a checkerboard with alternating black and white squares, using JavaFX
- Each square has a row and column index
- Let's use row-major order
 - access any square with `checkerboard[rowIndex][colIndex]`
- JavaFX `Rectangle`'s location can be set using row and column indices, multiplied by square width factor
 - row indicates Y values, column indicates X value

5 / 82

5

2D Arrays Example (2/2)

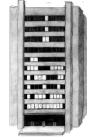
```
// instantiate a Pane and initialize the checkerboard 2D array
Pane myPane = new Pane();
Rectangle[][] checkerboard = new
    Rectangle[Constants.NUM_ROWS][Constants.NUM_COLS];
// loop through row and column indices (for each col in each row)
for (int row = 0; row < checkerboard.length; row++) {
    for (int col = 0; col < checkerboard[0].length; col++) {
        // instantiate rectangle, setting Y/X loc using row/col indices
        Rectangle rect = new Rectangle(col * Constants.SQ_WIDTH,
            row * Constants.SQ_WIDTH,
            Constants.SQ_WIDTH,
            Constants.SQ_WIDTH);
        // alternate black and white colors
        if ((row + col) % 2 == 0) {
            rect.setFill(Color.BLACK);
        } else {
            rect.setFill(Color.WHITE);
        }
        myPane.getChildren().add(rect); // graphically add the rectangle
        checkerboard[row][col] = rect; // logically add the rectangle
    }
}
```

6 / 82

6

SciLi Tetris: Loops and Arrays Writ Large

- In 2000, Tech House constructed then the largest Tetris game on the Scii – the Woz flew out to play it!
- 5 months of work: 11 custom-built circuit boards, a 12-story data network, a Linux PC, a radio-frequency video game controller, and over 10,000 Christmas lights – see <http://bastilleweb.techhouse.org/>
- Video: <https://www.youtube.com/watch?v=ikIRWoo8qrI&t=21s>
- Article: <http://news.bbc.co.uk/2/hi/science/nature/718009.stm>

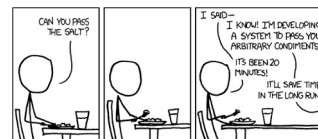


7 / 82

7

Lecture 14

Design Patterns and Principles: Part 1



From xkcd 103: https://xkcd.com/103/

8 / 82

8

“Design-Focused” Projects (1/2)

- Projects up to and including Fruit Ninja were considered “foundation-focused”
- Projects for remainder of semester are considered “design-focused”
 - given only an assignment specification (and hints), you will design programs from scratch
- On early projects, design was 25% of code grade; now 30-35%
 - for at least two of the following Fruit Ninja, Cartoon, Doodle Jump, Tetris, you will have Code Debriefs
 - 8% of your final grade is made up of Code Debriefs, where you will describe your design and code to TAs

9 / 82

9

“Design-Focused” Projects (2/2)

- Put much more effort (≥ 2 -3 hours) into understanding assignment specifications and planning **before coding**
 - containment/association and interface/inheritance diagrams crucial!
- Starting to code with a poor design leads to hours wasted trying to design and code on the fly

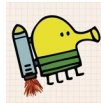


10 / 82

10

Design Grading

- Cartoon design grade will be based on design guidelines in the handout and discussed throughout this semester
 - will **NOT** be graded on specifics of this lecture
- Remaining projects' design **WILL** be graded with this week's design patterns + principles lectures in mind
 - refer to this lecture when designing DoodleJump with your partner!



11 / 82

11

Outline

- [Design in a Nutshell](#)
- [Abstraction and Encapsulation](#)
- [Class Cohesion and Coupling](#)
- [Wrapper Classes](#)

12 / 82

12

Context Beyond CS15

- Imagine you're working for a company with a bunch of software engineers that write the code for a popular app
- The app needs to work properly now, and in the future, more engineers will need to add new/change existing features
- Your job is to write code that:
 - works properly (*functionality*)
 - is easily readable (*style*)
 - another engineer can add to easily (*design*)
 - another engineer can modify easily (*design*)
- When writing real code, the **design** of your program is ultimately as important as its functionality

13 / 82

13

Design in a Nutshell (1/2)

- Up to now, focused on how to program
 - be appropriately lazy: re-use code and ideas
- Increasingly we learn about **good design**
- Some designs are better than others
 - "better" means, for example:
 - more efficient in space or time required (traditional criteria)
 - more robust, the "littles" – usability, maintainability, extensibility, scalability...
- These are central concerns of **Software Engineering**
 - discussed in detail in CS32 (CSCI0320)

14 / 82

14

Design in a Nutshell (2/2)



- There are trade-offs to make everywhere
 - architect balances aesthetics, functionality, cost
 - mechanical engineer balances manufacturability, strength, maintainability, cost
- Need to defend your trade-offs
 - no perfect solution, no exact rules
 - up to now designs rather straight-forward, not concerned about performance because not dealing with larger collections of data

15 / 82

15

What Do We Cover in These Lectures?

- Walk through process of planning design for a mock CS15 project
- Emphasize design principles and design patterns, which will be directly relevant to projects (including DoodleJump!),



16 / 82

16

Our Mock CS15 Project: Snake!

- Snake moves around board of squares at specified rate and continues moving in its last direction
- Player changes snake direction via key input, with goal of eating pellets to increase score
- Snake starts 3 squares in length, grows 1 square for each pellet eaten
- Snake can only move forward and turn right or left relative to its direction, not 180°
- Gain score by eating pellets – different colors yield different scores
- Game ends when snake moves off screen or into itself



CS15Snake Created by 2021 HTAs
Will Bunker and Harriet Mouchel © 17 / 82

17

Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes

18 / 82

18

Where do I start?!

- Assignment specifications can be daunting
- Start at highest level: brainstorm how to separate components of program (**delegation pattern!**)
 - containment/association decisions
 - what classes should we write? how should they communicate with each other?
 - critical to consider where to divide **abstractions**

19 / 82

19

Recall: Delegation Leads to Abstraction

- Delegation results in **levels of abstraction**, where each level deals with more specifics to complete an action



20 / 82

20

Abstractions (1/3)

- Each class represents an **abstraction**
 - a "black box": hides details that external users do not care about
 - allows you as the programmer to control programs' complexity – only think about relevant features



21 / 82

21

Abstractions (2/3)

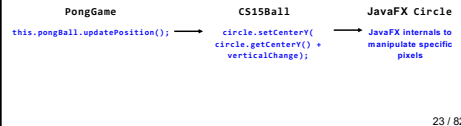
- CS15 support code and JavaFX are great examples of **levels of abstraction**



22

Abstractions (3/3)

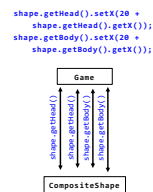
- CS15 support code itself results in **levels of abstraction**
 - each layer becomes more specific



23

Encapsulation(1/2)

- Lack of clean abstractions leads to messy communication between classes
- Example: `Game` class contains `CompositeShape` class that moves across screen
 - must allow access to `CompositeShape`'s private components via `getHead()` and `getBody()`
 - each communication between `Game` and `CompositeShape` internals is an arrow connecting them
- With access to those shapes, `Game` could also write code like `this.getHead().setFill(Color.RED);`
 - but what if we don't want `Game` to be allowed to change `CompositeShape`'s color?!



24

Encapsulation(2/2)

- We do this by...
 - delegating details to `CompositeShape`, simplifying communication
 - abstracting details of moving shapes, means no more need for `getHead()` and `getBody()`
 - so, `Game` doesn't need to know the details of moving shapes!
- Clean abstractions leads to clear communication between classes
- Key Point:** Use **getters/setters** ONLY as necessary to maximize encapsulation safety (IMPORTANT for future courses like CS200)
 - and you may well have a getter w/o a setter!

```

shape.moveRight();

Game
  |
  | shape.moveRight()
  |
CompositeShape

public void moveRight() {
    this.head.setX(20
    + this.head.getX());
    this.body.setX(20
    + this.body.getX());
}

```

25 / 82

25

Outline

- [Design in a Nutshell](#)
- [Abstraction and Encapsulation](#)
- [Class Cohesion and Coupling](#)
- [Wrapper Classes](#)

26 / 82

26

Review: Composition Pattern (1/3)

- You've used **composition** from the beginning
- Models object built through its containment of other objects and/or association with peer objects
- This is a **has-a** relationship, in which an object **has an** instance of another class stored as an instance variable
 - can be modeled through both **containment** – using the **new** keyword
 - as well as **association** – passing an object to an instance of another class to store as one of its instance variables
- Think of instance variables as modeling both the **components** and the properties/attributes that make up a class

27 / 82

27

Review: Composition Pattern (2/3)

- Compose one object out of other, more specialized objects that do one **specific** thing, e.g., car's engine
 - factor out code that works together for one specific purpose into a separate class (ex. `heatOven()` & `bakeCookies()` can go into a `Baker` class)
 - only instantiate an instance of this class if you need that functionality
 - specialist classes allow you to design components that you can build on
 - i.e., black boxes that expose only limited functionality
 - this is a form of delegation – don't rewrite code that specialists can do for you!
- Think of these specialist classes like Lego blocks that you can piece together to compose a larger class
 - every type of Lego block is unique and serves a **specific** purpose in your overall structure

28 / 82

28

Review: Composition Pattern (3/3)

- How can we determine good delegation and composition decisions?
- A `Car` class would use instances of these classes
 - `Engine`, `Brake`, `Transmission`, `SeatBelt`,....
 - `Car` can delegate `startUp()` to the `Engine`,...

29 / 82

29

High Cohesion and Loose Coupling (1/3)

- Cohesion refers to how well-defined the purpose of a single class is
- A class with a single, well-defined purpose has **high cohesion**
 - This is also known as the **Single Responsibility Principle**
- Strong **separation of concerns** reduces mental juggling – when coding in one class, only need to think about limited pieces of functionality – avoid "Swiss army knife" classes!
- You should be able to succinctly describe the purpose of each class in class header comments



30 / 82

30

High Cohesion or Low Cohesion?

High Cohesion	Low Cohesion
<ul style="list-style-type: none">• In a program modeling the life of a student, there is one CS15 class for the student to track their CS15 assignments• In Cartoon, one class that models a Cloud with 5 circles and moves each of the circles across the pane	<ul style="list-style-type: none">• In a program modeling the life of a student, there is one Life class that handles Fall classes, social life, and extracurriculars• In Cartoon, PaneOrganizer handles setting up the overall structure of panes, subpanes and shapes, and handles changing the color of each shape on key presses

31 / 82

31

High Cohesion and Loose Coupling (2/3)

- Coupling refers to how interdependent two classes are
- Each class should have **loose coupling** with other classes
 - use **abstractions** to keep clear relationships between classes
- Limit dependencies between classes.
 - should be able to modify internals of one class without worrying about impact on other classes

32 / 82

32

Coupling Example (1/3)

- Back to shape movement! Let's say we have our app to make a planet move via **Planet** class
 - to start, the planet is just represented by a **circle**

```
public class Planet {
    private Circle circle;
    public Planet() {
        this.circle = new Circle(Constants.PLANET_RADIUS);
    }
    public Circle getCircle() { return this.circle; }
}

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) ->
    venus.getCircle().setX(venus.getCircle().getX() + 10));
```

33 / 82

33

Coupling Example (2/3)

- Now we decide to use a composite shape with 4 rings around the planet
- First, move the **Circle** from **Planet** class
- Then, move the rings from **Planet** class
- Now every time a shape is added, it must be moved in **Cartoon**
- This is **tight coupling** (bad), i.e., **Cartoon** is too involved with details of moving **Planet**

```
// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) -> {
    venus.getCircle().setX(venus.getCircle().getX() + 10);
    venus.getRing1().setX(venus.getRing1().getX() + 10);
    venus.getRing2().setX(venus.getRing2().getX() + 10);
    // etc.
});
```

34 / 82

34

Coupling Example (3/3)

- Alternatively, could just have one **move** method in **Planet**
- Planet** could have 1 shape or 18 shapes, and **Cartoon** doesn't need to change!
- This is **loose coupling** (good)

```
public class Planet {
    // constructor and instance variables elided
    public void move() {
        // This method could move one shape or a bunch of shapes, but
        // Cartoon doesn't need to know about the details!
    }
}
```

```
// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1),
    (ActionEvent e) -> venus.move());
```

35 / 82

35

High Cohesion and Loose Coupling (3/3)

- Key Point:** Each class should have an independent, well-defined purpose (**high cohesion**), and communication between classes should be as simple and well-defined as possible (**loose coupling**)



36 / 82

36

TopHat Question

A **Tribute** class is using an instance of the **Bow** class to hunt for food. Which code in **Tribute** would indicate that the **Bow** class is written with proper encapsulation, abstractions, and loose coupling?

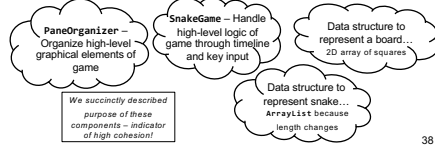
- A. `bow.getQuiver().getArrow().shoot();`
- B. `bow.shootArrow();`
- C. `bow.nockArrow("Wooden");`
`bow.drawBowString();`
`bow.looseArrow("Wooden");`

37 / 82

37

Back to Snake Brainstorming (1/3)

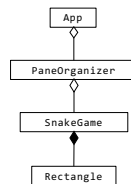
- Start at highest level: brainstorm how to separate components of program
 - keeping in mind aim of **high cohesion** and **loose coupling**



38 / 82

38

C/A Diagram Draft



39 / 82

39

Back to Snake Brainstorming (2/3)

- Let's think more about what's going on in the `SnakeGame` class
- What should happen at each tick of the `Timeline`?
- Let's write **pseudocode**:
 - move snake into next tile
 - if snake went off board or ran into itself:
 - game over
 - if pellet is on tile that snake moved into:
 - eat pellet
 - add to score
 - increase snake length by one square
 - generate new pellet



40 / 82

40

Back to Snake Brainstorming (3/3)

- We realize that each board square needs some extra information
 - is snake on the square?
 - is pellet on the square?
- With more complexity, let's consider delegating to a class `BoardSquare` rather than making `SnakeGame` handle it
 - instead of a board of "simple squares" (javaFX `Rectangles`), we need "smart squares" (our own `BoardSquare` class)
 - then we can model this extra information as **properties** (instance variables) of the square!

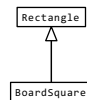
41 / 82

41

Designing the `BoardSquare` (1/3)

- Since each `BoardSquare` represents one graphical square, should we have `BoardSquare` inherit from a JavaFX `Rectangle`? Similar to a sports car inheriting from a car...

```
public class BoardSquare extends Rectangle { ...
```



42 / 82

42

Designing the BoardSquare (2/3)

- If `BoardSquare` extends `Rectangle`, `BoardSquare` inherits all of `Rectangle`'s methods
- That means `BoardSquare`'s set of public methods becomes the `Rectangle`'s set of public methods *plus* whatever specialized methods we write: is that a feature or a bug?
- In the context of Snake, we don't want programmers to have access to all `Rectangle` methods -- if they did, they could change position, size, rotation, etc. of `BoardSquare`



43 / 82

43

Designing the BoardSquare (3/3)

- **Key point:** To achieve simple communication between classes (**loose coupling**), the set of public methods a class or interface exposes should be as *simple* and *restricted* as possible
- Remember **encapsulation**... keep private parts your own business
- Let's only allow users of `BoardSquare` to access the limited parts we *need* to make public
- In this case, most of `Rectangle` methods shouldn't be accessible -- how can we make them private?

44 / 82

44

Outline

- [Design in a Nutshell](#)
- [Abstraction and Encapsulation](#)
- [Class Cohesion and Coupling](#)
- **[Wrapper Classes](#)**

45 / 82

45

Wrapper Classes

- A **wrapper** is code that **encapsulates** (or “wraps” around) another software component as a layer of **abstraction**
- In Java specifically, we create **wrapper classes** that add a layer of abstraction to another Java class
 - i.e., we add functionality to a class that other classes using it do not need to know details of
- Instead of inheriting from a class, **contain** an instance of that class as a component (in an instance variable)

46 / 82

46

BoardSquare Wrapper Class (1/2)

- **BoardSquare** wraps an instance of **Rectangle**
 - **Rectangle** is the main component of **BoardSquare**, but it also has extra functionality/information
- Allows us to restrict certain accesses inherited from **Rectangle** and add helpful pieces of information
 - **Pellet** contained in a **BoardSquare**
 - original **Color** of a **BoardSquare**

```
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, boolean odd) {
        this.square = new Rectangle();
        this.pellet = null;
        if (odd) {
            this.originalColor = Color.GREEN;
        } else {
            this.originalColor = Color.YELLOW;
        }
        this.setUpSquare(); // set size, location, ...
        pane.getChildren().add(this.square);
    }
}
```

47 / 82

47

BoardSquare Wrapper Class (2/2)

- A wrapper class exposes just the info that needs to be public and no more!
 - generally via setter and getter methods
- To show snake moving across board, one way is to change color of square to Black
 - so we add a setter for **Color**

```
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void setColor(Color color) {
        this.square.setFill(color);
    }
}
```

48 / 82

48

Keep Class Relationships Simple! (1/2)

- Is `setColor` the best we can do to abstract away internals of the square?
- For our game, we want:
 - square to turn black when snake goes over it
 - square to return to original color when snake moves off it
- With `setColor`, programmer could make square any arbitrary color – that shouldn't happen!

```
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void setColor(Color color) {
        this.square.setFill(color);
    }
}
```

49 / 82

49

Keep Class Relationships Simple! (2/2)

- Instead, let's have two separate methods
 - one method for snake moving onto square
 - one method for snake leaving square
- Trade-off: this produces more code but makes relationship between classes *simpler (looser coupling)*
- Key Point:** Strive for simpler class relationships – that may not always mean fewer methods!

```
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void addSnake() {
        this.square.setFill(Color.BLACK);
    }

    public void reset() {
        this.square.setFill(this.originalColor);
    }
}
```

50 / 82

50

Containment over Inheritance

- Wrapper classes are a good example of a generally agreed-upon design principle that containment is preferred to inheritance, unless the inheriting class should publicly expose all methods inherited.
- In our Snake example, our wrapper class is designed so `BoardSquare` has-a `Rectangle` as opposed to `BoardSquare` is-a `Rectangle`

51 / 82

51

TopHat Question

Which of the following is NOT true about wrapper classes?

- A. The goal of a wrapper class is to make a class's set of public methods as simple as possible
- B. Wrapper classes are an example of using encapsulation
- C. Wrapper classes add a layer of abstraction around some contained class
- D. Wrapper classes use inheritance rather than composition

52 / 82

52

Representing the Snake (1/2)

- Let's consider how to use `ArrayList` to represent the snake
- What should the `ArrayList` hold?
 - `BoardSquares` – hold whichever squares that snake is on top of
 - type will be `ArrayList<BoardSquare>`



53 / 82

53

Representing the Snake (2/2)

- `ArrayList` could be an instance variable in `SnakeGame` class... or could delegate it!
 - delegate for **higher cohesion**
- `Snake` class will act as **wrapper** class for `ArrayList<BoardSquare>` and *only* expose method to `move` and `changeDirection`
 - so much simpler than including all `Rectangle` methods
- **important note:** This decision means `SnakeGame` class won't have direct access to `ArrayList` so it can't mess with contents of list directly (**encapsulation!**)

54 / 82

54

Representing the Board

- We model our static board with a 2D array `BoardSquare[][]`
- Once board is created, the only editing to it will be to change state of individual `BoardSquare`
- Delegate to a `Board` class that acts as **wrapper** of `BoardSquare[][]`?
 - definitely **high cohesion** since `Board` would only handle board contents
 - no major benefit of delegating to `Board` as a wrapper since likely the only method would be a getter
 - `public BoardSquare tileAt(int row, int col)`
 - could argue for or against having separate `Board` class – both solutions are on GitHub!

55 / 82

55

Recap of Design Brainstorming So Far (1/2)

Class	Purpose	Important Instance Variables	Important Methods
App	Starts the application	n/a	n/a
PaneOrganizer	Organizes the high-level graphical organization of the program	BorderPane root	n/a
SnakeGame	Handles high-level logic of game via timeline and key input	Pane gamePane, Snake snake, Board board	updateGame (called on timeline), handleKeyInput (called on key press)

56 / 82

56

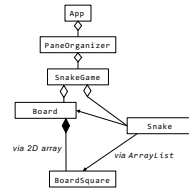
Recap of Design Brainstorming So Far (2/2)

Class	Purpose	Important Instance Variables	Important Methods
Snake	Represents snake moving around the board	<code>ArrayList<BoardSquare> snakeSquares,</code> <code>Board myBoard,</code> // to store association <code>Direction curDir</code> // Direction enum – Thu's lecture!	<code>move,</code> <code>changeDirection</code>
Board	Represents board of squares	<code>BoardSquare[][] board</code>	<code>tileAt</code>
BoardSquare	Represents one square on the board	<code>Rectangle boardSquare,</code> <code>Color originalColor,</code> <code>Pellet pellet</code>	<code>addSnake,</code> <code>reset,</code> <code>isleepy</code>

57 / 82

57

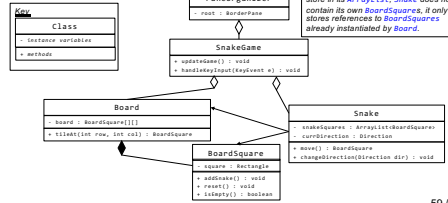
Containment/Association Diagram



58 / 82

58

Class Diagram



59 / 82

59

Announcements (1/2)

- [Snake code on GitHub](#) – check it out to see contrasting design decisions, and example of large program implementation
 - don't worry if some of it doesn't make sense, we will continue during Thursday's lecture
- 1D Arrays, ArrayLists, and Loops Section this week!
 - be sure to complete [mini-assignment](#) and send to section TAs prior to section
- DoodleJump Released Today!!
 - early handin: Monday 10/30
 - on-time handin: Wednesday 11/1
 - late handin: Friday 11/3
 - **do not underestimate this assignment! start early!**

60 / 82

60

Announcements (2/2)

- Code-Along: Debugging and GitHub
 - Wednesday October 25th
 - Sunday October 29th



61 / 82

61

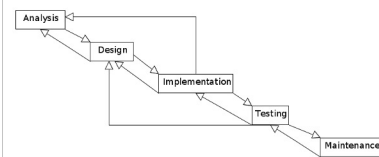
DoodleJump: Getting Started

- What classes should you represent in DoodleJump? What should their containment/association relationships be?
- How can you leverage "wrapper classes" to wrap some JavaFX elements you use to represent components of the program?
- How can you model properties like game score and doodle velocity? Which classes are those properties of?
- What do the different platforms have in common, and how are they different? How can you leverage polymorphism to make it so that the game doesn't need to know the actual type of each platform it moves?

62 / 82

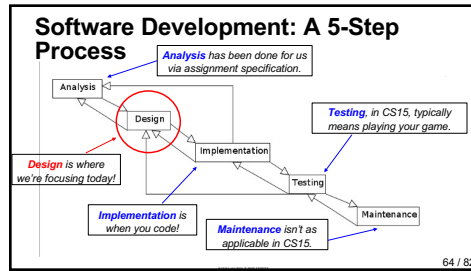
62

Software Development: A 5-Step Process

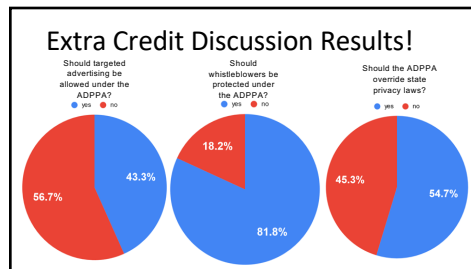


63 / 82

63



64




65



66

Surveillance Capitalism



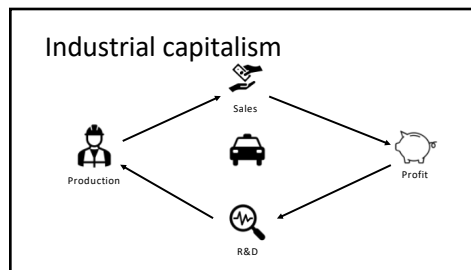
"I describe surveillance capitalism as the unilateral **claiming of private human experience as free raw material for translation into behavioral data. These data are then computed and packaged as prediction products and **sold into behavioral futures markets.**"**

— Shoshana Zuboff (retired Harvard Business School Professor and author of *The Age of Surveillance Capitalism*, 2019)

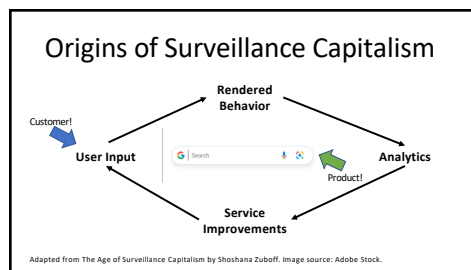
Surveillance capitalism is when companies gather our private information, analyze it, and then sell insights about our behavior to other businesses.

Image source: HBS

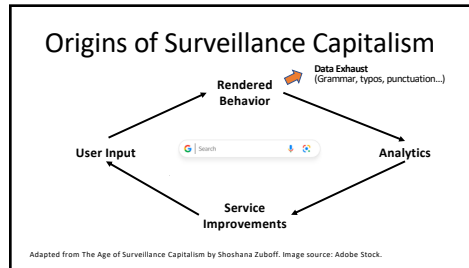
67



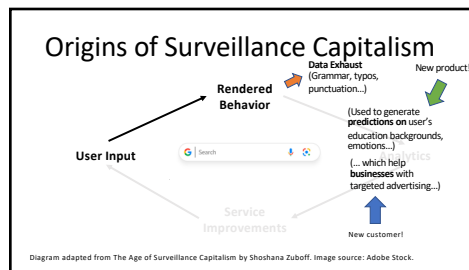
68



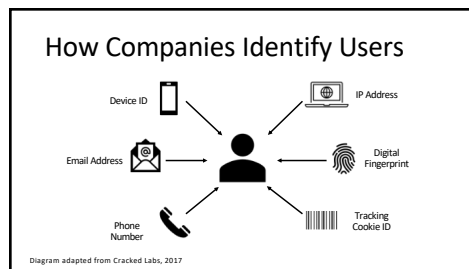
69



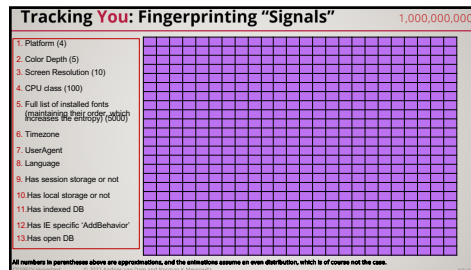
70



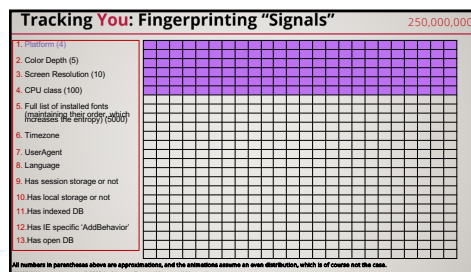
71



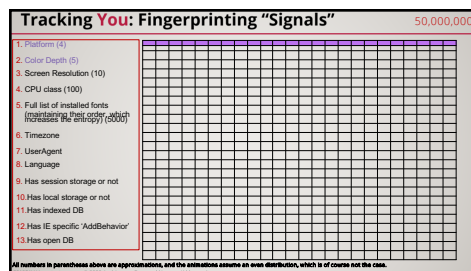
72



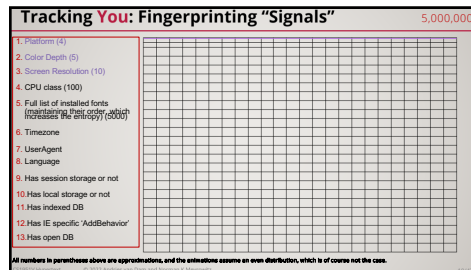
73



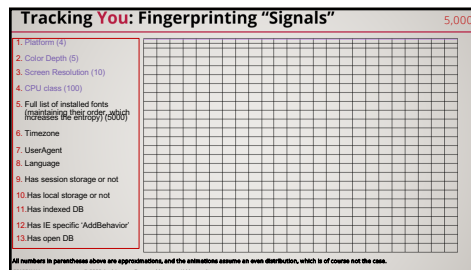
74



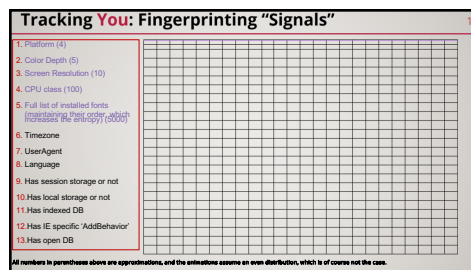
75



76

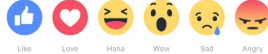


77



78

Incentive one: gather “better” data



Like Love Haha Wow Sad Angry

Image source: Meta

79

Incentive two: gather more data



More time users spend...
...more data collected...
...more accurate predictions and hence profit

Incentive for **addictive design** and **sensationalist content**

80

So what if accurate?



learn

the social dilemma NETFLIX

influence

Source: Twitter @BMJ1310000/2019

81



82



83
